頂点主体グラフ並列処理のための Giraph用クラスライブラリ

駒村 春野[1,a)]　岩崎 英哉[1,b)]

概要：本研究では，Giraph のプログラムにおいて現れる典型的な処理のパターンをデザインパターンとして抽出し，クラスライブラリを構築した．Giraph など頂点主体でグラフ並列処理を行うシステムでは，各頂点でスーパーステップと呼ばれる処理単位を同期を取りながら繰り返すことによって，並列計算を行う．しかし，グラフに対する処理は一般的にメッセージ送受信や集約処理により頂点の「計算状態」が遷移するため，各スーパーステップでは計算状態に応じた処理を行わなければならず，煩雑なプログラムになってしまう．本研究は，このような計算状態を陽に扱うグラフ処理の典型的なパターンをいくつか抽出し，Giraph のクラスライブラリとして構築した．このライブラリを使用することで，ユーザは，計算状態の遷移を意識することなくプログラムを記述することが出来る．いくつかの例題に関して，このライブラリを利用しなかった場合と利用した場合の 2 通りのプログラムを作成し，性能評価を行った結果も報告する．

キーワード：頂点主体，グラフ並列処理，デザインパターン，Giraph，クラスライブラリ

1. はじめに

グラフ並列分散処理システム Pregel [7] は，大規模なグラフを処理するための処理基盤である．Pregel は頂点単位で並列計算を行い，スーパーステップと呼ばれる処理単位を頂点間で同期をとりながら繰り返すことによって，並列計算を行う．プログラマは各頂点が行う処理を記述することで，プログラムを作成する．

スーパーステップ内では頂点間でデータの送受信を行うことができるが，送信されたデータは次のスーパーステップにならないと利用できない．また，グラフ全体の大域的な情報を取得するためには，**集約 (aggregation)** と呼ばれる処理を行う必要がある．集約も頂点間のデータ送受信を伴うため，集約する値を提出するときとグラフ全体で集約された結果を受け取るときとで，スーパーステップを分けなければならない．そのため，アルゴリズム中で頂点間通信や集約など，複数回の通信を行うときには，その度にスーパーステップを分断してプログラムを記述しなければならない．このようなプログラムは頂点に「状態」を保持させ，状態番号に基づく条件分岐で実装することができる．しかしながら，このようなプログラムの記述は煩雑で，また理解しにくいプログラムとなるという問題がある．

Pregel におけるグラフ問題を解くプログラムの実装では，類似した処理がよく見られる．そこで，類似した処理を持つ複数のプログラムの共通部分

[1] 電気通信大学 情報理工学研究科
[a)] komamura@ipl.cs.uec.ac.jp
[b)] iwasaki@cs.uec.ac.jp

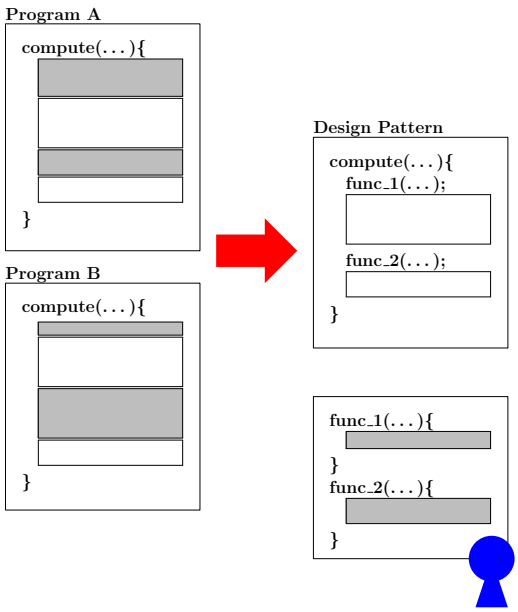

図 1　本研究の概要

を抜き出して，その内容を再利用することで，プログラム記述の手間を軽減することが出来ると考えられる．

以上を踏まえて本研究では，Pregel による頂点主体のプログラムにおいて共通して現れる典型的なパターンをデザインパターンとして抽出し，抽出したデザインパターンに対応するライブラリを提供する．プログラマは，このライブラリを利用することによって，より簡単・簡潔にプログラムを記述できるようになる．本研究は，Pregel の Java によるオープンソース実装のひとつである Giraph [1] を，ライブラリ実装の対象とする．

本研究では，まず複数のグラフアルゴリズムを Giraph を用いて実装し，プログラムに共通して現れる部分を見つけ，デザインパターンとして抽出する．共通する部分はライブラリに直接記述し，共通しない部分は抽象メソッドとすることで，デザインパターンライブラリを作成する．プログラマはこのライブラリを利用して各抽象メソッドを記述することで，求めるアルゴリズムを記述することができるようにする．

本研究の概要図を図 1 に示す．図では左の Program A，Program B から，共通の部分を白色の部分として，デザインパターンを抽出している．利用者は共通でない灰色の部分を記述することで，より少ない手間で求めるアルゴリズムを記述することができる．

性能実験として，本ライブラリを利用したことによる記述量の変化，オーバーヘッドを確かめるため，ライブラリを使用した場合と使用していない場合で記述量・実行時間・スーパーステップ数等を測定，比較した．

本稿の構成は次の通りである．まず 2 節で Pregel の概要，3 節で Giraph の提供するクラスについて述べる．4 節では抽出されたパターン，5 節で抽出したパターンをもとにライブラリの実装について説明する．6 節で実装したライブラリの評価実験を行う．7 節で実験結果の考察を行う．8 節では関連研究について述べる．9 節で本稿をまとめる．

2. Pregel

頂点主体グラフ並列分散処理システム Pregel [7] は，大規模なグラフを処理するための処理基盤である．頂点主体の処理によって，幅広いグラフアルゴリズムを柔軟に表現することができる．

2.1 バルク同期並列

Pregel における処理はバルク同期並列 (Bulk Synchronous Parallel, BSP) [10] の概念に基づいている．

Pregel の処理では，開始時に，グラフの各頂点が計算ノード上のワーカに割り振られ，それぞれのワーカ上で計算が行われる．計算はスーパーステップと呼ばれる処理の繰り返しで行われる．各スーパーステップにおいて，各頂点は次の処理を行うことができる．

- 前回のスーパーステップで他頂点から送信された値の受信
- 自身の計算処理
- 他頂点への値の送信
- 集約子への値の提出
- 集約子からの値の取得

全頂点は各スーパーステップの終了時に同期して，次のスーパーステップに処理を移す．

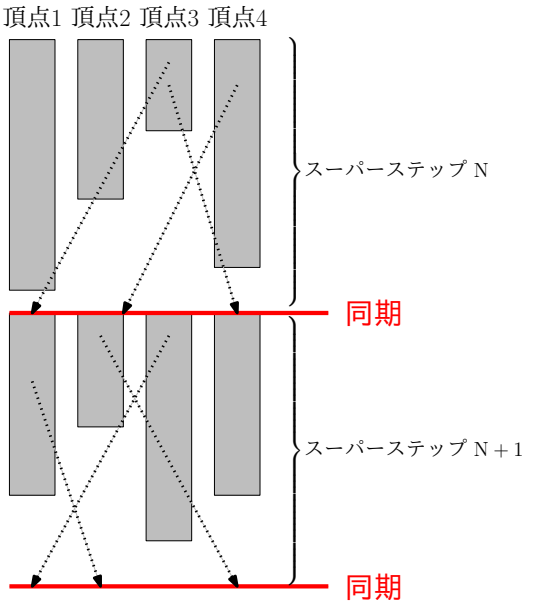

図 2　バルク同期並列

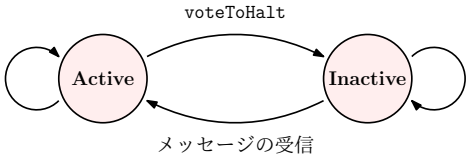

図 3　頂点の状態を示すオートマトン ([7] より引用)

```
1  class SimpleShortestPathsComputation {
2    compute(v, messages) {
3      if (superstep() == 0) {
4        v.value = Double.MAX_VALUE;
5        if (v.id == 0) {
6          v.value = 0;
7          for (e : v.edges)
8            sendMessage(e.targetId, e.value);
9        }
10     } else {
11       minDist = v.value;
12       for (m : messages)
13         minDist = Math.min(minDist, m);
14       if (minDist < v.value) {
15         v.value = minDist;
16         for (e : v.edges) {
17           distance = minDist + e.value;
18           sendMessage(e.targetId, distance);
19         }
20       }
21     }
22     vertex.voteToHalt();
23   }
24 }
```

図 4　Giraph における単一始点最短経路問題の擬似コード

図 2 にバルク同期並列の概要図を示す．図中灰色の長方形は各計算ノードでの処理，点線はメッセージを表す．

Pregel の実装は非公開であるが，オープンソース実装として Apache Giraph [1]，Pregel+ [11]，Apache Hama [2]，GPS [8] などが存在する．

2.2　処理の流れ

各スーパーステップでは各頂点で `compute` メソッドを同期的に実行することで並列計算を行う．

計算の終了は，各頂点の停止投票 (`voteToHalt`) に基づいている．各頂点の状態は活動中または停止中のどちらかにあり，`compute` 内の処理で状態が遷移する．頂点の状態を示すオートマトンを図 3 に示す．

プログラムの開始時，つまりスーパーステップが 0 のとき，すべての頂点は活動中状態である．あるスーパーステップ内で計算に参加して `compute` メソッドを実行するのは，すべての活動中状態の頂点である．停止中状態の頂点では，スーパーステップでの処理を行わない．各頂点は `voteToHalt` メソッドを呼ぶことによって，自分自身を停止中状態にすることができる．停止中の頂点は，他の頂点からメッセージを受信すれば，再び活動中状態に遷移する．

プログラムの実行は，すべての頂点が同時に停止中，かつ未受信のメッセージがない場合に終了する．

2.3　プログラム例：単一始点最短経路問題

図 4 に Pregel のオープンソース実装の Giraph における単一始点最短経路問題の擬似コードを示す．このプログラムは，指定された番号の頂点を開始頂点として，全頂点に対する最短経路を求める．

入力グラフの各辺は非負の重みを距離として持ち，辿った辺の重みの総和が少ないものが最短経

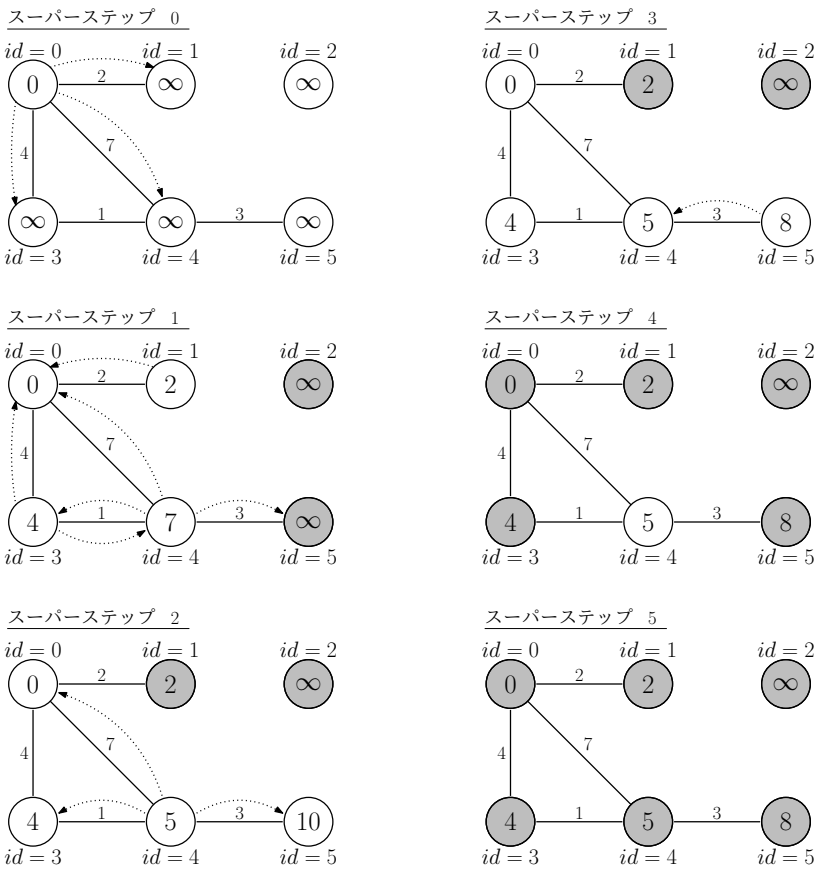

図 5 単一始点最短経路問題の例

路となる．開始頂点から辺を辿って辿りつけないような頂点に対しては，重みを無限大とする．このプログラムの処理によって，最終的に各頂点が保持する値は開始頂点から自身の頂点までの最短経路の距離となる．

Giraph では，`compute` メソッドをインスタンスメソッドとして持つクラスを定義する．引数 `v`，`message` はそれぞれ `compute` メソッドを実行する頂点と，隣接頂点から送られてきたメッセージを意味する．

まずスーパーステップ 0 では各頂点は自身の値 (`v.value`) を 0 (開始頂点の場合) か ∞ (それ以外の場合) に設定し，隣接頂点へ頂点の値に辺の重みを加えた値を送信する．各スーパーステップでは隣接頂点から送られてきたメッセージの中で，最小の値を頂点自身の持つ値と比較して小さい方を新しい頂点の値とする．頂点の値が更新された場合，隣接頂点に対して，頂点の値と隣接頂点へ接続する辺の重みの合計をメッセージとして送信する．最後に，`voteToHalt` を呼び出して，メッセージが送られてこない場合にはそれ以上処理を続けないようにする．

始点からの距離と隣接頂点への距離の和は，始点から隣接頂点までの距離になるため，送られたメッセージの最小値を求めることで，始点からの最短距離を求めることが出来る．

図 5 に，図 4 の単一始点最短経路問題の擬似コードを適用した計算の過程を示す．点線はメッセージを，白色の頂点が活動中状態を，灰色の頂点は停止中状態を表す．

3. Giraph

Giraph は 2 節で説明した Pregel の Java によるオープンソース実装である．本節では，Giraph の機能について説明する．本研究で提供するライブラリは，Giraph で定義されたクラスのサブクラスを作成することで実装している．ここでは，Giraph プログラムの処理の流れと本研究に関連するクラスについて説明する．

3.1 処理の流れ

一般的な Giraph プログラムの処理は以下から構成される．

(1) 入力を読み込み，グラフを初期化する．
(2) 計算が終了するまで，スーパーステップを繰り返す．
(3) 最終的なグラフを出力する．

Giraph では各頂点で実行するスーパーステップ 1 回分の内容を `compute` というメソッドに記述することで，プログラムを作成する．

3.2 `Computation` クラス

`Computation` クラスは，各スーパーステップにおいて各頂点で実行される `compute` メソッドを記述するためのクラスである．プログラマは `Computation` クラスを継承したクラスを定義し，`compute` メソッドをオーバーライドして記述することで，求めるプログラムを実装する．

`compute` メソッド内では，次のことが可能である．

- 頂点の値，外向辺 (頂点が持つ辺) の値の変更
- 前回のスーパーステップで送られたメッセージの受信
- 他の頂点へのメッセージの送信 (送信したメッセージは次のスーパーステップで受け取られる．)
- 前回のスーパーステップで集約子に提出された値の受信
- 集約子への値の提出

はじめの操作によって，グラフの形が変化することもある．

3.3 `Aggregator` クラス

`Aggregator` クラスは，集約子の機能を実現するためのクラスである．

集約子はすべての，または一部の頂点の情報を集約するための機能であり，各頂点は任意のスーパーステップで集約子へ値を提出することができる．提出された値は，+ 等の簡約演算子を用いて結合される．結合した結果の値は，次のスーパーステップで，全頂点で利用できるようになる．Giraph には，min，max，sum のような複数の組み込み集約子が存在する．

集約子はグラフの大域的な情報の収集や統計のために使用することができる．例えば，各頂点の出次数を合計すれば，有向グラフの辺の総数を求めることができる．

集約子はプログラム全体を整合させるためにもよく用いられる．例えば，入力値の論理積を値として持つ and 集約子を利用することで，すべての頂点が条件を満たしているかを判断することができる．また，mim または max 集約子を用いて頂点番号を集約することによって，プログラムにおいて，他の頂点とは異なった，特別な役割を行う頂点を選ぶことができる．

新しい集約子を定義するためには，まず既存の `Aggregator` クラスのサブクラスを作り，集約子の値が最初の入力によってどう初期化されるのか，どのようにして集約された複数の値を一つの値に簡約するのかを定義する．

3.4 `MasterCompute` クラス

`MasterCompute` クラスでは各スーパーステップの開始時に行う下記の master compute の内容を記述できるほか，プログラムの開始時に行う内容など，計算全体に関する処理を行うことができる．master compute は GPS [5,8] によって導入され，Giraph にも実装されている独自の拡張機能である．プログラマは `MasterCompute` クラスを継承したクラスを定義し，それぞれのメソッドをオーバーライドして内容を記述する．

master compute 内では次のことなどを行うことができる．

- 集約子の登録，値の変更
- 計算の強制終了

MasterCompute クラスはオプションであるが，3.3 節の集約子を利用する場合などには必ず定義しないといけない．

3.5 Writable クラス

Writable クラスは Giraph においてデータを扱うためのクラスである．

Giraph では，頂点や辺の値，そして集約子によって集約される値などすべてに対して Writable クラスを用いる．そのため，頂点や辺が複数の値を持つような場合には，複数の変数を内包した Writable クラスを定義する必要がある．

3.6 プログラム例：最大値問題

簡単なプログラムの例として，2.3 節で説明した単一始点最短経路問題に加えて，最大値問題を説明する．

図 6 に最大値問題の Giraph プログラムを示す．このプログラムはグラフ中の連結成分の頂点が保持する値の最大値を求める．

このプログラムの処理によって，最終的に各頂点が保持する値は最大の値と同じ値となる．

まずスーパーステップ 0 では自身の値を隣接頂点へ送信する．各スーパーステップでは隣接頂点から送られてきたメッセージの中で，最大の値を頂点自身の持つ値と比較して大きい方を新しい頂点の値とする．頂点の値が更新された場合，その値を隣接頂点に対してメッセージとして送信する．最後に，voteToHalt をして，メッセージが送られてこない場合にはそれ以上処理を続けないようにする．

この処理の繰り返しによって，最終的に各頂点の値は連結成分中の最大値となる．

4. パターン

本節は本研究で提案するパターンの概要を説明する．

```
1 class MaxValue {
2   compute(v, messages) {
3     if (getSuperstep() == 0) {
4       sendMessageToAllEdges(v, v.value);
5     } else {
6       maxValue = v.value;
7       for (m : messages)
8         maxValue = Integer.max(maxValue, m);
9       if (v.value < maxValue) {
10        v.value = maxValue;
11        sendMessageToAllEdges(v, v.value);
12      }
13    }
14    vertex.voteToHalt();
15  }
16 }
```

図 6 最大値問題

アルゴリズム 1 追跡パターン

```
Require: G = (V, E)
 1: for v ∈ S in parallel do
 2:     M ← 受信したメッセージ
 3:     d ← digest(M)
 4:     if update(v.val, d) then
 5:         v.val ← computeNewVal(v.val, d)
 6:         for e ∈ E do
 7:             m ← computeMessage(v, e)
 8:             sendMassage(e.targetId, m)
 9:         end for
10:     end if
11:     voteToHalt
12: end for
```

4.1 追跡パターン

追跡パターンは，グラフの辺を辿っていくことで解決する問題に適応できるパターンである．3.6 節で示した最大値問題と単一始点最短経路問題はこのパターンに属する．

追跡パターンでは，すでに訪問した頂点から出力辺を辿り隣接頂点へメッセージを送信し，その隣接頂点を訪問する．

追跡パターンのアルゴリズムをアルゴリズム 1 に示す．下線で示した部分はそれぞれのプログラムで変更可能な部分である．アルゴリズム中の各関数の内容は以下のようになっている．

- digest(M)

アルゴリズム 2 Densest Subgraph [3]

Require: $G = (V, E)$ and $\epsilon > 0$
1: $\bar{S}, S \leftarrow V$
2: **while** $S \neq \emptyset$ **do**
3: $A \leftarrow \{v \in S \mid \deg_s(v) \leq 2(1+\epsilon)f(S)\}$
4: $S \leftarrow S \setminus A$
5: **if** $f(S) > f(\bar{S})$ **then** $\bar{S} \leftarrow S$ **end if**
6: **end while**
7: **return** $\bar{S}$

送られてきたメッセージをそれぞれのアルゴリズムによる方法で 1 つの値にまとめる．

- update($v.val, d$)
  頂点自身の値を更新するかどうか判断する．
- computeNewVal($v.val, d$)
  新しい頂点の値を計算する．
- computeMessage(v, e)
  隣接頂点に送るメッセージの値を計算する．

4.2 減少パターン

減少パターンは，与えられた条件を満たす部分グラフを求める問題に適応できるパターンである．出力は入力で与えられたグラフの頂点集合の部分集合 S であり，問題の解は S による誘導部分グラフとなる．

減少パターンでは，S が全ての頂点を含んでいる状態から始め，与えられた条件を満足しない頂点を S から順次取り除くことで，求める部分グラフを導く．頂点が脱落するごとに，その時点でのグラフ S を評価関数によって評価し，最も良い評価関数値のグラフを解とする．

このパターンに属するアルゴリズムとしては，与えられたグラフの密な部分グラフを求める Densest Subgraph [3]，GreedyOQC [9] が挙げられる．アルゴリズム 2 に Densest Subgraph のアルゴリズム，アルゴリズム 3 に GreedyOQC のアルゴリズムをそれぞれ示す．ここで，$f(S)$ は各アルゴリズムにおけるグラフの密の程度を表す評価関数であり，具体的には次のように定義される．

- Densest Subgraph: $f(S) = |E(S)|/|S|$
- GreedyOQC: $f(S) = |E(S)| - \alpha_{|S|}C_2$

アルゴリズム 2，3 より，共通部分を抽出して，アルゴリズム 4 に示すような減少パターンを得る．

アルゴリズム 3 GreedyOQC [9]

Require: $G = (V, E)$
1: $\bar{S}, S \leftarrow V$
2: **while** $S \neq \emptyset$ **do**
3: $u \leftarrow \arg\min_{v \in S} deg\{v\}$
4: $S \leftarrow S \setminus \{u\}$
5: **if** $f(S) > f(\bar{S})$ **then** $\bar{S} \leftarrow S$ **end if**
6: **end while**
7: **return** $\bar{S}$

アルゴリズム 4 減少パターン

Require: $G = (V, E)$
$\tilde{S}, S \leftarrow V$
while $S \neq \emptyset$ **do**
for $v \in S$ **in parallel do**
if judge(v) **then** $S \leftarrow S \setminus \{v\}$ **end if**
if $f(S) > f(\tilde{S})$ **then** $\tilde{S} \leftarrow S$ **end if**
end for
end while
return $\bar{S}$

各頂点 v は，部分グラフ S に自身が残るべきかどうかを judge(v) によって判断し，残るべきでないと判断した場合には自分自身を S から取り除く．実際のパターンの実装においては，自身を `voteToHalt` によって休止させることにより，S からの脱落を実装するので，減少パターンでは活動中状態の頂点がその時点で S に含まれる頂点になる．

5. ライブラリの実装

5.1 クラス構成

5.1.1 基本クラス

各デザインパターンのライブラリは，基本的に以下の 3 つのクラスから構成される．

- Computation クラス
  各スーパーステップで実行する内容を表す．
- VertexWritable クラス
  頂点の持つデータを表す．
- InputFormat クラス
  入力形式を表す．本ライブラリでは頂点が複数の変数を持つため，入力をどの変数に読み込むかを定義する．

5.1.2 頂点の状態

他の頂点から送信されたメッセージ，集約子によって集約された情報は，次のスーパーステップにならないと取得することができない．そのため，メッセージ通信や集約子でしか得られない情報を利用するためには，スーパーステップを分割し，実質的な反復処理一回分を複数の (N 回とする) スーパーステップで構成する必要がある．

これを実現する素朴な方法は，compute メソッド内でスーパーステップ数を N で割った余りによって，処理の内容を分岐させることである．例えば，$N = 2$ のときは，スーパーステップ数が偶数の場合と奇数の場合でそれぞれ異なる処理を行うようにする．

しかし本ライブラリでは，メッセージ送信や集約子による情報の集約が必要でない場合には，スーパーステップを分割させずに処理を進めることが可能となるように上で述べた方法は採らない．そのかわり，頂点に状態を表す変数 state を持たせて state が持つ値に従って，スーパーステップ内の処理を分岐させる．

集約子を用いる場合などスーパーステップの分割が必要な場合，現スーパーステップでの処理を一度終わらせて，次のスーパーステップで次の状態の処理を行うが，スーパーステップの分割が必要ない場合には同じスーパーステップにおいて次の状態の処理を引き続き行うようにする．

5.1.3 Aggr クラス

各ライブラリにおける集約操作を抽象化する目的で Aggr クラスを定義した．Aggr クラスは Aggregator クラスのラッパクラスである．

通常，Computation クラスのインスタンスメソッドである，値の集約を行うメソッドを Aggr クラスのインスタンスメソッドとすることで，各アルゴリズムにおいて集約子の扱うクラスの違いを意識することなく実装することができる．

標準的に用いるクラスとして，次のような Aggr クラスの子クラスを定義している．

- IntSumAggr クラス
  集約された整数値の合計値を求める．
- IntIntSumAggr クラス
  集約された整数値のペアのそれぞれの合計値を求める．
- EdgeNumAggr クラス
  辺の数の合計を求める．
- VertexEdgeNumAggr クラス
  全頂点数と辺の数の合計を集約する．
- NullAggr クラス
  何も集約しない (何も集約する必要が無い場合に用いる)．

これらのクラスは aggregate メソッドを実装している．NullAggr 以外のクラスではこのメソッドは実際に値を集約して真を返し，NullAggr クラスでは何もせずに偽を返す．したがって，aggregate メソッドの返り値を調べることで，実際に集約が行われたか，すなわちスーパーステップをおわらせる必要があるか否かを判断できる．

5.2 追跡パターン

図 7 に追跡パターンの compute メソッドの擬似コードを示す．また，図 8 に追跡パターンにおける頂点の状態遷移図を示す．

追跡パターンでは，各頂点 v は，状態を表す変数 state，出力辺を表す edges (edges は Giraph が用意している変数である) 以外に次の変数を持つ．

- val: 自身の値を表す．
- newval: 自身の次の (新しい) 値を表す．

また，追跡パターンの Computation クラスはインスタンス変数として，Aggr の子クラスのインスタンス aggr を持つ．aggr は全頂点での値が求まる前に処理を取りやめる場合に用いる．

ユーザが実装するメソッドは以下の 5 つである．

(1) initVertex(v)
  頂点 v を初期化する．

(2) computeMessage(v, e, a)
  頂点 v が辺 e で隣接する頂点に対して送信するメッセージを計算する．ここで，a は集約子から得られた値である．

(3) initNewValue()
  新たな頂点の値 newv の初期値を返す．(newv は computeNewValue(newval, val) によって次々と更新される．)

```
1 class TracingPatternComputation {
2   computeSS(v, messages) {
3     boolean f = true;
4     switch (v.state) {
5       case 0:
6         initVertex(v);
7         for (e : v.edges) {
8           m = computeMessage(v, e, null);
9           sendMessage(e.targetId, m);
10        }
11        v.state = 1;
12        v.voteToHalt();
13        return;
14      case 1:
15        newv = initNewValue();
16        for (m : messages)
17          newv = computeNewValue(newv, m);
18        newv = computeNewValue(newv, v.val);
19        v.newval = newv;
20        f = aggr.aggregate(this, v);
21        v.state = 2;
22        if (f) return;
23      case 2:
24        a = aggr.getAggregatedValue(this);
25        if (computeFlag(a)) {
26          v.voteToHalt();
27          return;
28        }
29        if (v.val != v.newval) {
30          v.val = v.newval;
31          for (e: v.edges) {
32            m = computeMessage(v, e, a);
33            sendMessage(e.targetId, m);
34          }
35        }
36        v.state = 1;
37        if (inactivateVertices)
38          v.voteToHalt();
39    }
40  }
41 }
```

図 7　追跡パターン

(4) computeNewValue(newval, val)
　newval と val から新しい値を計算する．
(5) computeFlag(a)
　これ以上処理を続けるかどうか判断する．

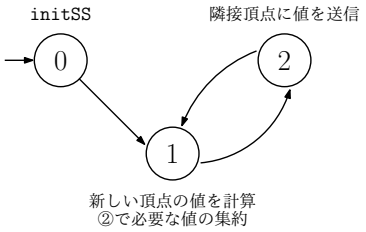

図 8　追跡パターンにおける頂点の状態遷移図

また，以下の変数を定義した．

- inactiveVertices
　処理中に頂点を停止中状態にするかどうかを真偽値で示す変数

5.3　減少パターン

プログラム 9 に減少パターンの compute メソッドの擬似コードを示す．また，図 10 に減少パターンにおける頂点の状態遷移図を示す．

減少パターンでは，各頂点 v は，状態を表す変数 state，出力辺を表す edges 以外に次の変数を持つ．

- sTilde: 自身が最終的な解を示す頂点集合 $\tilde{S}$ に含まれているかどうかを表す．解は最も大きな sTilde を持つ頂点の集合となる．
- inS: 自身が頂点集合 S に含まれているかどうかを表す．
- degree: 自身の外向辺のうち，S に含まれる頂点に向かうものの数を表す．

また，減少パターンの Computation クラスはインスタンス変数として，Aggr の子クラスのインスタンス aggr1，aggr2 を持つ．aggr1 は評価関数値の計算に必要な値を（あれば）集約によって求める必要がある場合に用いる．また，aggr2 は脱落するかどうかの判断のために用いる値を集約によって求める必要がある場合に利用する．

ユーザが実装するメソッドは以下の 3 つである．

(1) computeEvalValue(v, aggr)
　部分グラフの評価関数を計算する．
(2) computeDropout(v, aggr)
　頂点が脱落するべきかどうかを真偽値で返す．
(3) computeVertex(v)
　評価値に応じて解を更新する．

```
1 class DecreasePattern {
2   computeSS(v, messages) {
3     boolean f = true;
4     switch (v.state) {
5       case 0:
6         initVertex(v);
7         v.state = 1;
8       case 1:
9         receiveDropoutMessage();
10        f = aggr1.aggregate();
11        v.state = 2;
12        if (f) return;
13      case 2:
14        if (f) computeEvalValue(v, aggr1);
15        f = aggr2.aggregate();
16        v.state = 3;
17        if (f) return;
18      case 3:
19        if (f) aggr2.aggregate();
20        if (computeDropout(v, aggr2)) {
21          sendDropoutMessage(v);
22          v.inS = false;
23          v.state = 4;
24          v.voteToHalt();
25          return;
26        }
27        computeVertex(vertex);
28        v.state = 1;
29        return;
30      case 4:
31        v.voteToHalt();
32        return;
33    }
34  }
35 }
```

図 9　減少パターン

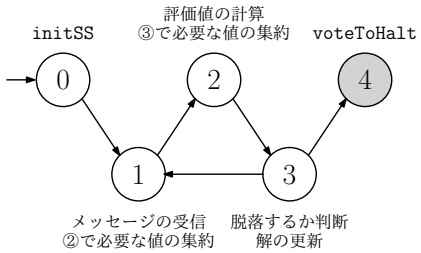

図 10　減少パターンにおける頂点の状態遷移図

表 1　実験環境

OS	Debian GNU/Linux (wheezy)
CPU	AMD Opteron Processor 6380 x4 (16x4 core)
Memory	128GB (DDR3-1600)
Storage	360GB HDD SATA3
Hadoop	1.2.1
Giraph	1.2.0

また，減少パターン内で予め以下の関数を用意している．

(1) `sendDropoutMessage(v)`
　隣接頂点へ自身が脱落する旨のメッセージを送信する．

(2) `receiveDropoutMessage()`
　隣接頂点から送信された脱落する旨のメッセージを受信し，該当頂点へ接続する辺を削除する．

6. 評価実験

本ライブラリを利用したプログラムの性能を評価するため，プログラムの記述量，処理に必要になるスーパーステップ数，実行時間の 3 つについて計測し，手書きで記述したプログラムと比較した．本実験を行った環境を表 1 に示す．

6.1 入力データ

本実験の入力データにはランダムに作成した頂点数 10K, 辺の数 1M の無向グラフを用いる．無向グラフの辺は，辺を結ぶ頂点間においてそれぞれが始点，終点となるような，両方向の 2 つの有向辺を持つことによって表現する．

6.2 アルゴリズム

実験に使用したアルゴリズムを表 2 に示す．

また，これまでに言及が無いアルゴリズムに対して説明する．

Reachability
　指定した頂点から辺を辿って，他の頂点へ到達することができるかどうかを全頂点に対して判定する．

N Reachability

表 2　実験に使用するアルゴリズム

	名前	説明
追跡パターン	Simple Shortest Path	単一始点最短経路問題 (2.3 節)
	Max Value	最大値問題 (3.6 節)
	Reachability	到達可能性問題
	N Reachability	N 頂点到達可能性問題
減少パターン	Densest Subgraph	最密部分グラフ問題 [3]
	GreedyOQC	最密部分グラフ問題 [9]

Reachability に似ているが，到達可能な頂点が N 個以上見つかった時点で即座に計算を終える．

6.3 プログラムの記述量

ライブラリ未使用時，使用時の `Computation` クラスの記述量を比較する．また，ライブラリの `Computation` クラスの記述量も同様に，以下の 3 点を比較する．

- ライブラリ未使用の場合
- ライブラリ使用の場合
- ライブラリとライブラリ使用の場合の合計

コメントを除いた，純粋な処理について記述したプログラム量の，行数とバイト数を計測する．

追跡パターンの記述量を表 3，減少パターンの記述量を表 4 に示す．各アルゴリズムに対して，上の行がライブラリ未使用のとき，下の行がライブラリを使用したときの記述量になっている．

6.4 スーパーステップ数

処理が終了するまでに実行したスーパーステップの数を計測する．

追跡パターンのスーパーステップ数を表 5，減少パターンのスーパーステップ数を表 6 に示す．

6.5 実行時間

処理が終了するまでに経過した時間を計測する．Hadoop におけるプログラムの実行結果は，Web UI で確認することができる．Web UI で表示される，Giraph Timer の Total を実行時間として参照し，5 回実行した平均を計測値とする．また，ワーカタスクの数が 8, 16 の 2 通りの場合について計測を行った．GreedyOQC については，実行時間が非常に大きくなるため，8 コアのみの測定とした．

表 3　追跡パターン: 記述量

	行	バイト	合計 行	合計 バイト
Simple Shortest Path				
ユーザ記述	32	1,218		
ユーザ記述	30	1,098	101	3,508
ライブラリ本体	71	2,410		
Max Value				
ユーザ記述	20	725		
ユーザ記述	28	866	99	3,276
ライブラリ本体	71	2,410		
Reachability				
ユーザ記述	28	1,023		
ユーザ記述	30	968	101	3,378
ライブラリ本体	71	2,410		
N Reachability				
ユーザ記述	34	1,358		
ユーザ記述	41	1,219	112	3,629
ライブラリ本体	71	2,410		

追跡パターンの実行時間を表 7，減少パターンの実行時間を表 8 に示す．

7. 考察

ライブラリを使用した場合のプログラムのバイト数について，減少パターンについては最大およそ 24% 程度の削減が見られた．追跡パターンについても，ほとんどのプログラムで 5% から 10% の削減がみられた．各メソッド名と引数の記述がオーバーヘッドとなっているため，実際のプログラム

表 4　減少パターン: 記述量

	行	バイト	合計 行	合計 バイト
Densest Subgraph				
ユーザ記述	38	1,491		
ユーザ記述	37	1,372	111	3,889
ライブラリ本体	74	2,517		
GreedyOQC				
ユーザ記述	45	1,928		
ユーザ記述	39	1,467	113	3,984
ライブラリ本体	74	2,517		

表 5　追跡パターン: スーパーステップ数

	未使用	使用
Max Value	5	5
Reachability	5	5

表 6　減少パターン: スーパーステップ数

	未使用	使用
Densest Subgraph	19	19
GreedyOQC	30,001	30,001

表 7　追跡パターン: 実行時間 (単位:s)

アルゴリズム	ライブラリ	8 コア	16 コア
Max Value	未使用	10.95	11.59
	使用	11.03	11.57
Reachability	未使用	10.79	11.30
	使用	11.11	11.77

表 8　減少パターン: 実行時間 (単位:s)

アルゴリズム	ライブラリ	8 コア	16 コア
Densest Subgraph	未使用	11.93	13.01
	使用	11.91	12.77
GreedyOQC	未使用	54.01	–
	使用	54.38	–

の記述量はより少なくなっていると考えられる．

記述量が増加している Reachability について，実装したアルゴリズムが単純なものであるため，もともとの記述量も少なく，このオーバーヘッドが顕著に見られたのだと考えられる．また，デザインパターンにより汎用性を持たせるため，恣意的に冗長に記述している部分もあり，このことも記述量を増加させる原因となっている．

実行時間について，スーパーステップ数の変わらない Max Value などから，ライブラリの冗長部分によるオーバーヘッドはほとんどないと考えられる．

スーパーステップ数はそれぞれのプログラムで変化は見られず，ライブラリを適用しても冗長なスーパーステップがないことが確認できる．

8. 関連研究

8.1 Filtering

Filtering [6] は MapReduce を用いたグラフアルゴリズムのデザイン手法である．MapReduce における Reduce フェイズとその次の Map フェイズを結合して 1 つのラウンドとして扱い，各ラウンドで何をするのか設計する．Pregel ではなく MapReduce の概念を用いて実装を行うため，効率的なアルゴリズムは設計できても，効率的に記述が出来ているとは言い難い．また，MapReduce でグラフを扱うためには，グラフデータ全てを通信する必要があるため，大きなオーバーヘッドが発生する可能性がある．

8.2 Fregel

Fregel [4] は関数型頂点主体のグラフ処理を記述するためのドメイン固有言語 (Domain-Specific Language, DSL) である．本研究と比較すると，煩雑な Pregel プログラミングを軽減する点が共通しているが，手段が異なる．Fregel は DSL であり，処理を関数プログラムとして記述して，Giraph へコンパイルすることで Giraph プログラムを作成する．一方，本研究ではアルゴリズムの共通部分を抜き出してライブラリ化することで，比較的簡単に Giraph プログラムを作成することを目的とする．

9. おわりに

本稿では，複数のグラフアルゴリズムから共通するパターンを抜き出すことによって，Giraph で効率的に記述するためのライブラリを作成した．作成したライブラリでは抽象クラスを用いて，提

案パターンに当てはまる幅広いアルゴリズムを，そのアルゴリズムに固有の処理のみ追加で記述することにより実現できる．本研究では減少パターン，追跡パターンの 2 つのパターンを抽出・実装することができた．今後の課題としては，本ライブラリをより直感的に利用できるように，ユーザ記述部の範囲を考察すること，そして，さらに汎用的なパターンを抜き出すことが挙げられる．

謝辞 本論文に対するコメントを下さった京都大学の馬谷誠ニ助教，国立情報学研究所の対馬かなえ助教に深く感謝いたします．

参考文献

[1] Apache Giraph: . `http://giraph.apache.org/`.

[2] Apache Hama: . `http://hama.apache.org/`.

[3] Bahmani, B., Kumar, R. and Vassilvitskii, S.: Densest subgraph in streaming and mapreduce, *Proceedings of the VLDB Endowment*, Vol. 5, No. 5, pp. 454–465 (2012).

[4] Emoto, K., Matsuzaki, K., Hu, Z., Morihata, A. and Iwasaki, H.: Think like a vertex, behave like a function! a functional DSL for vertex-centric big graph processing, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ACM, pp. 200–213 (2016).

[5] GPS: . `http://infolab.stanford.edu/gps/`.

[6] Lattanzi, S., Moseley, B., Suri, S. and Vassilvitskii, S.: Filtering: a method for solving graph problems in mapreduce, *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, ACM, pp. 85–94 (2011).

[7] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N. and Czajkowski, G.: Pregel: a system for large-scale graph processing, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ACM, pp. 135–146 (2010).

[8] Salihoglu, S. and Widom, J.: Gps: A graph processing system, *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ACM, p. 22 (2013).

[9] Tsourakakis, C., Bonchi, F., Gionis, A., Gullo, F. and Tsiarli, M.: Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees, *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, pp. 104–112 (2013).

[10] Valiant, L. G.: A bridging model for parallel computation, *Communications of the ACM*, Vol. 33, No. 8, pp. 103–111 (1990).

[11] Yan, D., Cheng, J., Lu, Y. and Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation, *Proceedings of the 24th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, pp. 1307–1317 (2015).